論文 / 著書情報
Article / Book Information

Title	A Low-Storage-ConsumptionXML Labeling Method forEfficient Structural Information Extraction
Author	Wenxin Liang, Akihiro Takahashi, Haruo Yokota
Journal/Book name	Database and Expert Systems Applications of LNCS (Proc. of DEXA2009), ， ，
発行日 / Issue date	2009，9
DOI	http://dx.doi.org/10.1007/978-3-642-03573-9_2
権利情報 / Copyright	The original publication is available at www.springerlink.com.
Note	このファイルは著者（最終）版です。 This file is author (final) version.

A Low-Storage-Consumption XML Labeling Method for Efficient Structural Information Extraction

Wenxin Liang[1⋆], Akihiro Takahashi[2⋆⋆], and Haruo Yokota[2,3]

[1] School of Software, Dalian University of Technology
[2] Department of Computer Science, Tokyo Institute of Technology
[3] Global Scientific Information and Computing Center, Tokyo Institute of Technology
liang@computer.org, akihiro@de.cs.titech.ac.jp, yokota@cs.titech.ac.jp

Abstract. Recently, labeling methods to extract and reconstruct the structural information of XML data, which are important for many applications such as XPath query and keyword search, are becoming more attractive. To achieve efficient structural information extraction, in this paper we propose C-DO-VLEI code, a novel update-friendly bit-vector encoding scheme, based on register-length bit operations combining with the properties of Dewey Order numbers, which are not able to be implemented in other relevant existing schemes such as ORDPATH. Meanwhile, the proposed method also achieves lower storage consumption because it does not require either prefix schema or any reserved codes for node insertion. We performed experiments to evaluate and compare the performance and storage consumption of the proposed method with those of the ORDPATH method. Experimental results show that the execution times for extracting depth information and parent node labels using the C-DO-VLEI code are about 25% and 15% less, respectively, and the average label size using the C-DO-VLEI code is about 24% smaller, comparing with ORDPATH.

1 Introduction

The Extensible Markup Language (XML) has rapidly become the *de facto* standard for representing and exchanging data on the Internet, because it is portable for representing different types of data from multiple sources. An XML document can be modeled as a nested tree structure in which each element is treated as a node with a start tag and end tag. An example XML document and its corresponding XML tree are shown in Figure 1 and Figure 2, respectively.

To handle the hierarchical tree model of XML data, XML labeling methods are required to extract and reconstruct the structural information of the XML document, which are commonly used for many applications such as XPath

⋆ This work was done when the author was with Japan Science and Technology Agency (JST) and Tokyo Institute of Technology.

⋆⋆ The author is currently with NTT Data Corporation.

```
<BOOK ISBN = "1-55860-438-3">
 <SECTION>
  <TITLE>Bad Bugs</TITLE>
   Nobody loves bad bugs
  <FIGURE CAPTION="Sample bug" />
 </SECTION>
 <SECTION>
  <TITLE>Tree Frogs</TITLE>
   All right-thinking people
  <BOLD>love</BOLD>free frogs.
 </SECTION>
</BOOK>
```

Fig. 1. Example XML document

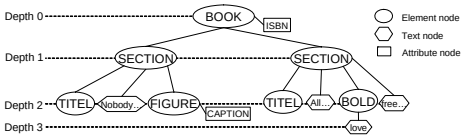

Fig. 2. Example XML tree

query [19] and keyword search [16]. XML labeling methods assign a label to each node of the XML tree, and each labeled node is stored as a tuple containing the tag information for element nodes or the Parsible Character Data values of text nodes together with the label. Structural information, such as containment relationship, order of sibling and depth of nodes in the XML tree, is possible to be obtained from the regularity of the labels.

A number of XML labeling methods based on numbering schemes such as the preorder–postorder [10] and Dewey Order (DO) [19] have been proposed. The preorder–postorder method assigns preorder and postorder numbers to all nodes to maintain the containment and order information between nodes. The DO method uses a delimiter to separate the label of a parent node from the code that expresses the sibling order among its children. The parent–child and ancestor–descendant relationships, and the relative sibling order, can be deduced by comparing the labels at corresponding positions determined by the delimiters. The DO method is effective for obtaining the labels of parent and ancestor nodes and for determining common ancestors of multiple nodes by using the delimiters. These properties of the DO method are useful for achieving efficient XPath query [19] and keyword search [16]. For example, the parent, ancestor and depth information can be directly determined by analyzing the delimiters, which is valuable to enable efficient XPath axis determination in XPath query and efficient LCA or SLCA extraction in keyword search. However, using simple numerical values for the sibling order in the preorder–postorder and DO methods has the problem of expensive update costs. If a new node is inserted in an intermediate position of the XML tree, many node labels have to be renumbered to maintain the proper sibling order.

To address the update problem, several update-friendly XML labeling methods such as DO Variable-Length-Endless-Insertable (DO-VLEI) code [13] and ORDPATH [17] have been proposed. The DO-VLEI labeling method inherits features of the DO method, but reduces the update cost for insertion operations. It uses the VLEI code [13] for expressing the sibling order by a unique magnitude relationship, which enables the unlimited insertion of new nodes with no relabeling of other nodes being required. ORDPATH, which is implemented in Microsoft® SQL Server^{TM} 2005, is also an update-friendly labeling method based on the DO number. [18] reported that ORDPATH achieves both better structural information extraction and storage consumption than the existing prefix-based labeling methods such as LSDX [8] and persistent labeling schemes [9, 12]. DO-VLEI

achieves efficient structural information extraction as ORDPATH does. However, the size of DO-VLEI codes becomes larger when numbering large-scale XML documents.

Other two update-friendly labeling schemes, namely QED [14] and CDBS [15], are more efficient than ORDPATH in respect to data updates. However, QED works worse in some cases and CDBS results in both worse structural information extraction and storage consumption than ORDPATH. Therefore, it is essential to have both efficient structural information extraction and low storage consumption and for large-scale XML data in update-friendly methods. The ORDPATH method uses a Compressed binary representation (C-ORDPATH) and a prefix schema to reduce the size of node labels and to achieve effective structural information extraction. However, as criticized in MonetDB/XQuery [5, 6], C-ORDPATH has problems of expensive manipulation and storage costs. On the structural information extraction point of view, the features of C-ORDPATH cause the problem of high manipulation cost because decode of C-ORDPATH requires to traverse from the head through the whole code and refer to the prefix schema. With respect to the storage consumption, the label in the C-ORDPATH cannot be optimized during the initial labeling, because negative integers and even numbers are reserved for node insertions. In addition, the labels available for assignment to the XML nodes are limited by the prefix schema. Therefore, the prefix schema used for C-ORDPATH must be changed when handling large XML documents, which causes further expensive manipulation and storage costs.

To achieve both efficient structural information extraction and low storage-consumption for large XML documents with high update frequency, in this paper we propose the Compressed-bit-string DO-VLEI (C-DO-VLEI) code, an update-friendly XML labeling method. To achieve efficient structural information extraction, we propose a novel and efficient method based on register-length bit operations combining with the properties of DO numbers, which are not able to be implemented in other bit-vector encoding schemes including ORDPATH. Meanwhile, C-DO-VLEI also achieves lower storage consumption by using a compressed binary representation using DO numbering schemes without any prefix schema. We performed experiments to evaluate and compare the storage consumption and performance of the proposed method with those of the C-ORDPATH method. Experimental results show that the proposed method outperforms the ORDPATH method with respect to both structural information extraction and storage consumption, particularly for documents of large size and depth.

The main contributions of this paper can be listed as follows.

1. We propose a novel update-friendly bit-vector encoding scheme, C-DO-VLEI code, for efficiently extracting the structural information between nodes from the C-DO-VLEI code. The proposed method enables efficient structural information extraction based on register-length bit operations combining with the properties of DO numbers instead of bit-by-bit comparisons utilized in the C-ORDPATH method.

2. The C-DO-VLEI code is also able to shorten the label length and thereby reduce the storage consumption required for the labeling of large-scale XML documents, because it does not require either prefix schema or any reserved codes for node insertion.
3. We performed experiments to compare the performance and storage consumption of the proposed method with those of C-ORDPATH which is verified in [18] that it outperforms other existing methods in both storage consumption and performance. Experimental results indicate that: 1) the execution times for extracting depth information and parent node labels using the C-DO-VLEI code are about 25% and 15% less, respectively, than for C-ORDPATH, which enables the proposed method to achieve high performance in such applications as XPath query and keyword search; 2) the label size using the C-DO-VLEI code is about 24% smaller than that using C-ORDPATH, which means taht the proposed method also outperforms the ORDPATH method with respect to storage consumption.

The remainder of the paper is organized as follows. In Section 2, we briefly introduce related work. Section 3 proposes the C-DO-VLEI code. Section 4 describes the method for extracting structural information from the C-DO-VLEI code. Section 5 describes the experiments and the evaluation of the proposed methods. Finally, Section 6 concludes the paper.

2 Related Work

One traditional labeling method is the preorder–postorder method [10], which assigns preorder and postorder numbers to all nodes. Another traditional method is the DO method [19], which uses a delimiter "." to separate a parent label from its child's label. However, when simple consecutive integers are used for node labeling in these methods, many nodes must be relabeled to satisfy sibling order following the insertion of a new node into an intermediate position.

To tackle this problem, several methods have been proposed. The Quartering-Regions Scheme [4] uses floating-point numbers with the preorder–postorder method. Range labeling methods [7] prepare appropriate intervals between preorder–postorder labels in advance. However, these methods still require relabeling of many nodes when the prepared intervals for node insertions are used up. MonetDB is an open-source database system for high-performance applications in data mining, OLAP, GIS, XML query, text and multimedia retrieval [1]. MonetDB/XQuery [5, 6] encodes XML documents based on the preorder–postorder method but achieves reasonable update costs by using a technique based on logical pages. However, MonetDB/XQuery still cannot completely avoid renumbering the *size* and *level* fields of some nodes when updating the document. Therefore, it is predicted that the update costs will degrade the performance of MonetDB/XQuery, especially when processing large-scale XML documents with high update frequency.

Labeling methods that are capable of unrestrictedly inserting nodes without relabeling any nodes, have been proposed [13–15, 17, 20]. In [20], the label of a

newly inserted node is made from the product of a prime number and the label of its parent node. The label of the ancestor node can be extracted by factorizing the label of the child node. However, the factorization is still quite expensive. In [18], it was reported that the ORDPATH method [17] achieves both better structural information extraction and storage consumption than the existing prefix-based labeling methods such as LSDX [8] and persistent labeling schemes [9,12]. Other two update-friendly labeling schemes, namely QED [14] and CDBS [15], are more efficient than the ORDPATH method in respect to data updates. However, QED has worse performance in some cases and CDBS results in both worse structural information extraction and storage consumption than the ORDPATH method. The DO-VLEI code [13] can achieve efficient structural information extraction as the ORDPATH method does. However, the size of DO-VLEI codes becomes larger when labeling large-scale XML documents. The DO-VLEI code [13], which is the basis of our proposed method, and the ORDPATH method, which is the baseline of comparison in our experiments, will be described in detail in the following two subsections.

2.1 DO-VLEI Code

We first introduce the definition of the VLEI (Variable Length Endless Insertable) code as follows.

Definition 1 (VLEI Code). *A bit string $v = 1 \cdot \{0|1\}^*$is a VLEI code, if the following condition is satisfied.*

$$v \cdot 0 \cdot \{0|1\}^* < v < v \cdot 1 \cdot \{0|1\}^*$$

For example, $10 < 1 < 11$ and $100 < 10 < 101 < 1 < 110 < 11 < 111$. A new VLEI code can be unrestrictedly generated from two arbitrary adjacent VLEI codes by an effective algorithm [13]. The VLEI code can be used for both the preorder–postorder and the DO methods. Here, we focus on the combination of the DO method and the VLEI code. It is named DO-VLEI code and defined as follows.

Definition 2 (DO-VLEI Code).
1. The DO-VLEI code of the root node $C_{root} = 1$.
2. The DO-VLEI code of a non-root node $C = C_{parent} \cdot C_{child}$, where C_{parent} denotes the DO-VLEI code of its parent and C_{child} denotes the VLEI code satisfying the appropriate sibling order.

The DO-VLEI code of the nth ancestor node can be obtained by extracting the character string between the leftmost character and the nth delimiter. An example XML document tree labeled by DO-VLEI code is illustrated in Figure 3.

2.2 ORDPATH

ORDPATH [17] is an update-friendly XML labeling method implemented in Microsoft® SQL Server^{TM} 2005, in which arbitrary node insertions require no

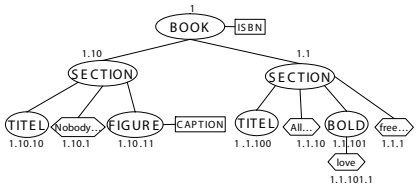

Fig. 3. Labeling by DO-VLEI code

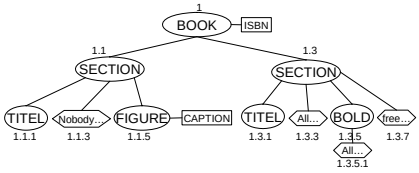

Fig. 4. Labeling by ORDPATH

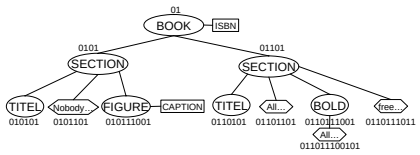

Fig. 5. Labeling by C-ORDPATH

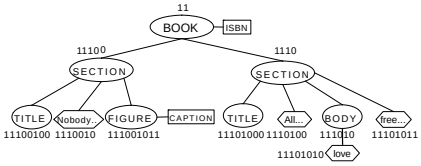

Fig. 6. Labeling by C-DO-VLEI code

relabeling of the existing nodes. The ORDPATH labels are also based on the DO method, in that a child label is made from a sibling code, a delimiter, and the label of its parent node. Figure 4 shows an XML tree labeled by ORDPATH. In ORDPATH, only positive, odd numbers are assigned for the initial labeling; negative integers and even numbers are reserved for insertions. The end of a sibling code must be an odd number. For example, when the ORDPATH label of a node is newly inserted between nodes labeled "1.3.1" and "1.3.3", the label "1.3.2.1", which is made by adding "1" to "1.3.2", is assigned to the node. The sibling code of "1.3.2.1" is "2.1".

ORDPATH is implemented by a compressed binary representation using bit strings $\{0, 1\}$, and is called C-ORDPATH. A pair of bit strings L_i and O_i are used to represent an integer by using a prefix schema (see reference [17] for details). An ORDPATH label is a string of delimited integers, and i is the order of the integer within the string. L_i is a bit string specifying the number of bits of O_i. L_i is specified by analysis of the tree using the prefix schema. O_i is treated as a binary number within a range set by L_i. For example, ORDPATH "1.5" is represented in C-ORDPATH as (0111001). (0111001) is divided into (01,110-01) by sequential analysis using the prefix schema in [17]. (01) and (110-01) are converted into "1" and "5" based on the O_i value range.C-ORDPATH represents delimiters by delimiting the bit string according to the prefix schema and compresses the label size. An XML tree labeled by the C-ORDPATH is shown in Figure 5.

3 C-DO-VLEI Code

In this section, we propose a compressed binary representation of the DO-VLEI code, called C-DO-VLEI code, which aims to reduce the label size. In this paper,

“” denotes a character string, and () denotes a bit string. For example, “01” represents a concatenation of character *0* and *1*, and (01) represents a bit sequence of bit *0* and *1*.

3.1 Components of the DO-VLEI Code

A DO-VLEI label is a variable-length character string constructed from the three characters “.”, “1”, and “0”, which satisfies the following three conditions.

(1) Consecutive “.” characters do not appear.
(2) “.” does not exist at the tail end of a label.
(3) The VLEI code starts with “1”.

According to the above conditions, the “1” that is the beginning of a VLEI code always appears after “.”. Therefore, “.” and “1” can be combined as “.1”, meaning that a DO-VLEI code is composed of the three elements “.1”, “1”, and “0”.

3.2 Definition of C-DO-VLEI Code

The C-DO-VLEI code is defined as follows.

Definition 3 (C-DO-VLEI Code). *The three elements of a DO-VLEI code: “.1”, “1”, and “0”, are represented by the bit strings: (10), (11), and (0), respectively. This compressed binary representation of DO-VLEI codes is called C-DO-VLEI code.*

Note that the shortest bit string (0) is assigned to the “0” that can be adjusted to appear more frequently than the other two components, and that the set of bit strings: (0), (10), and (11) are prefix codes. A prefix code is a set of words such that no word of the set is a prefix of another word in the set. By sequentially analyzing the bit string from the prefix code, C-DO-VLEI codes can be uniquely decoded into the original DO-VLEI codes. Figure 6 shows an XML tree labeled by C-DO-VLEI codes.

4 Structural Information Extraction

In this section, we propose an efficient method, based on register-length bit operations combining with the properties of Dewey Order numbers, for extracting structural information between nodes from the C-DO-VLEI code. The extracted structural information, such as depth, parent information, are necessary and valuable for determining XPath axes in XPath query and for detecting LCAs or SLCAs in XML keyword search. However, due to the space limitation, here we take XPath query as an example to show how the extracted structural information work to determine the XPath axes. The most important and frequently used XPath axes are listed in Table 1, in which parent, preceding-sibling, child and

Table 1. XPath axes

Axis	Description
ancestor	all the ancestors of the context node (parent, grandparent, etc.)
preceding	all the nodes that precede the context node in the document except any ancestor nodes
descendant	all the descendants of the context node (children, grandchildren, etc.)
following	all the nodes that appear after the context node except any descendant nodes
parent	the single node that is the parent of the context node
preceding-sibling	all the nodes that have the same parent as the context node and appear before the context node
child	the children of the context node
following-sibling	all the nodes that have the same parent as the context node and appear after the context node

following-sibling are subsets of ancestor, preceding, descendant, and following, respectively.

Next, we explain how to obtain this information from the C-DO-VLEI codes by using the properties of DO and simple code comparisons.

4.1 Using Properties of DO

Since C-DO-VLEI codes use the DO method, information about 1) depth of node, 2) parent's label and 3) ancestors' labels can be obtained by the following three operations that use the properties of DO, respectively.

DO-1: Count the number of delimiters.

DO-2: Extract the prefix code before the rightmost delimiter.

DO-3: Repeat ***DO-2*** until the last delimiter is reached.

It is important to find the locations and count the number of the delimiters in C-DO-VLEI to implement ***DO-1***–***DO-3***. The most straightforward method is to decode the label from the beginning to the end to detect the bit string (10) that is assigned for the delimiter. However, this is expensive when handling long codes. To reduce the processing cost, we propose an efficient method for detecting the delimiters by using register-length bit operations instead of bit-by-bit comparison.

Since the delimiter in C-DO-VLEI codes is assigned to bit string (10), any place where the bit changes from (1) to (0) is a delimiter candidate. However, when "10" appears in the DO-VLEI code, the corresponding C-DO-VLEI will be (110). That is, the last two bits are also (10). In order to distinguish these two patterns of (10), we focus on the number of consecutive (1s) before the (0). In the fragment of C-DO-VLEI code shown in Figure 7, wherever an odd number of consecutive (1s) appears before (0), the last two bits (10) represent a delimiter, while (10) ending at an even number of consecutive (1s) represents the DO-VLEI code "10". Therefore, it is possible to detect the delimiters by counting the number of consecutive (1s). Next, we describe algorithms for operations ***DO-1***–***DO-3*** based on delimiter detection. Notations for the bit operations used in the proposed algorithms are shown in Table 2.

Algorithm for *DO-1* The operation ***DO-1***, namely the operation for detecting the node depth, can be implemented by the following steps.

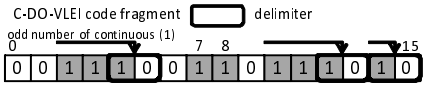

Fig. 7. Example fragment of a C-DO-VLEI code

Table 2. Bit operation notation

notation	description
$x \ll y$	left shift
$x \overset{u}{\gg} y$	right shift (logical shift)
$\overline{x}$	NOT
$x \& y$	AND
$x \mid y$	OR

D-1: Generate a bit string, *endPointOfOne*, in which the initial values of all the bits are (0), and only the last bit at the place where an odd number of consecutive (1s) appears is set to (1). For the C-DO-VLEI code v shown in Figure 4.1, the bit string *endPointOfOne* can be generated by $v \& \overline{v \ll 1}$.

D-2: Generate a bit string *endPointOfOne'*, in which the initial values of all the bits are (0), and only the bit at the place where (0) changes to (1) in the C-DO-VLEI code is set to (1). For the C-DO-VLEI code v shown in Figure 4.1, the bit string *endPointOfOne'* can be generated by $endPointOfOne \overset{u}{\gg} 1$.

D-3: Assuming that each bit of the code is assigned an integer ID, as shown in Figure 4.1, then the bit (1) in *endPointOfOne* is a delimiter if the condition $ID(i) - ID(j) = oddnumber$ is satisfied, where $ID(i)$ and $ID(j)$ denote the ID of (1) in *endPointOfOne* and its nearest leftmost (1) in *endPointOfOne'*, respectively. For example, the $5th$ bit (ID=4) in *endPointOfOne* is a delimiter because the ID of its nearest leftmost (1) in *endPointOfOne'* is 1. However, the $9th$ bit (ID=8) in *endPointOfOne* is not a delimiter because the ID of its nearest leftmost (1) *endPointOfOne'* is 6. Algorithm 1 shows the details of delimiter detection. Algorithm 1 outputs a bit string *pointOfDelimiter*, in which the initial values of all the bits are (0), and only the bit indicating the place of delimiters is set to (1).

D-4: After the delimiters are found by Algorithm 1, the node depth can be determined by counting the number of (1s) in *pointOfDelimiter*. Algorithm 2 shows the details for counting the number of (1s) in a bit string based on the divide-and-conquer algorithm [11].

Algorithm for *DO-2* and *DO-3* ***DO-2*** finds the last delimiter from *pointOfDelimiter* generated by Algorithm 1, which is implemented by the following steps.

P-1: Generate *pointOfDelimiter*.

P-2: Generate a *mask* by
$\overline{(\text{pointOfDelimiter}\& - \text{pointOfDelimiter}) - 1 \mid label}$, as shown in Figure 9.

Algorithm 1 Delimiter detection

Input: C-DO-VLEI code v
Output: pointOfDelimiter
1: endPointOfOne $\leftarrow v \& \overline{v \ll 1}$
2: delimiterEven $\leftarrow \overline{v} \& (v +$ (endPointOfOne&0x5555))&0xAAAA
3: $v' \leftarrow v \overset{u}{\gg} 1$
4: endPointOfOne $\leftarrow$ endPointOfOne$\overset{u}{\gg}1$
5: delimiterOdd $\leftarrow \overline{v'} \& (v' +$ (endPointOfOne $\leftarrow$ 0x5555))&0xAAAA
6: pointOfDelimiter $\leftarrow$ delimiterEven|delimiterOdd

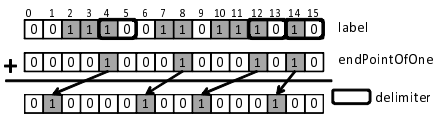

Fig. 8. Example of delimiter detection

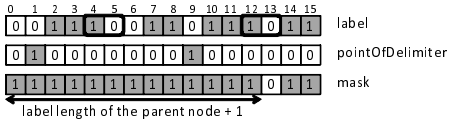

Fig. 9. Example of mask for extracting parent label

P-3: The length of the parent label L_p can be obtained by counting the number of consecutive (1s) from the head of the *mask*. Algorithm 3 shows the details of how to count the number of (1s) from the head of a C-DO-VLEI code. Finally, the parent label can be generated by outputting the L_p bits from the head of the original label.

In addition, ***DO-3***, namely obtaining the ancestors' labels, can be implemented by repeating the above operations.

4.2 Using Code Comparison

Theorem 1. *Let the bit strings v_0, v_d, and v_1 represent the three elements of the C-DO-VLEI code "0", ".1" and "1", respectively. According to Definition 3, $v_0 = (0)$, $v_d = (10)$ and $v_1 = (11)$. Using Algorithm 4, comparison of the C-DO-VLEI codes for two nodes yields the proper document order if v_0, v_d and v_1 satisfy the following equation*[4].

Algorithm 2 Depth detection

Input: v (16-bit unsigned integer)
Output: number of (1) in v (depth)
1: $v \leftarrow v - ((v \overset{u}{\gg} 1) \& 0x5555)$
2: $v \leftarrow (v \& 0x3333) + ((v \overset{u}{\gg} 2) \& 0x3333)$
3: $v \leftarrow ((v + (v \overset{u}{\gg} 4)) \& 0x0F0F)$
4: $v \leftarrow v + (v \overset{u}{\gg} 8)$
5: $v \leftarrow v + (v \overset{u}{\gg} 16)$
6: depth $\leftarrow v \& 0x001F$
7: **return** depth

[4] Here we do not give the proofs of this theorem and the other relevant ones due to the space limitation.

Algorithm 3 Count number of consecutive (1s) from the head of code

Input: v (16-bit unsigned integer)
Output: length of parent node, L_p
1: $v \leftarrow \overline{v}$
2: $n \leftarrow 16$
3: $c \leftarrow 8$
4: **repeat**
5: $x \leftarrow v \overset{u}{\gg} c$
6: **if** $x \neq 0$ **then**
7: $n \leftarrow n - c$
8: $v \leftarrow x$
9: **end if**
10: $c \leftarrow c \overset{u}{\gg} 1$
11: **until** $c \neq 0$
12: **return** $L_p = n - x$

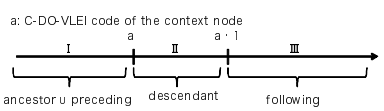

Fig. 10. Code range for ancestor, preceding, descendant and following

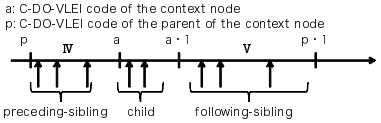

Fig. 11. Code range for parent, preceding-sibling, child and following-sibling

$$1 \cdot v_0 < 1 < 1 \cdot v_d < 1 \cdot v_1 \tag{1}$$

Moreover, according to Definition 3 and the conditions described in Section 3.1, the code b for a descendant node of the node with code a is $b = a \cdot 10 \cdot \{11|0\}^*$. Therefore, according to Definition 1, $a \cdot 10 \cdot \{0\}^* \leq b < a \cdot 11 \cdot \{0\}^*$. That is, the codes of a and b satisfy the following equation, where $\prec_{CDV}$ denotes the comparison based on Algorithm 4.

$$a \prec_{CDV} b \prec_{CDV} a \cdot 1 \tag{2}$$

Similarly, the ancestor, preceding, and following nodes can be also obtained by the code comparison using Algorithm 4. Therefore, as shown in Figure 10, using the code a of a context node and $a \cdot 1$ can divide the nodes into three ranges **I**, **II** and **III** for ancestor and preceding nodes, descendant nodes, and following nodes, respectively.

Algorithm 4 Code comparison

Input: two C-DO-VLEI codes, v,w
(length(v)≤length(w))
Output: $v' \geq w$ is true or false
1: $l' \leftarrow length(w) - length(v)$
2: $v' \leftarrow v \cdot 1 \cdot \{0\}^{l'-1}$
3: **if** $v' \geq w$ **then**
4: **return true**
5: **else**
6: **return false**
7: **end if**

Table 3. XPath axis extraction method

Axis	Extraction method
ancestor	***DO-3***
preceding	(node-set extracted from range **I**) - ancestor
descendant	node-set extracted from range **II**
following	node-set extracted from range **III**
parent	***DO-2***
preceding-sibling	nodes ∈ range **IV** ∩ nodes with the same depth as the context node (***DO-1***)
child	nodes ∈ range **II** ∩ nodes whose depth is one more than the context node (***DO-1***)
following-sibling	nodes ∈ range **V** ∩ nodes with the same depth as the context node (***DO-1***)

Table 4. Experimental Environment

CPU:	Dual-Core Intel Xeon 5110 (1.60GHz)
Memory:	DDR2 ECC FB-DIMM 9GB(2GB×4, 512MB×2)
Storage:	D-RAID RAID 0+1
HDD:	SEAGATE ST3750640AS (750GB, 7200rpm, 3.5inch)
OS:	Linux 2.6.18 CentOS 5(Final)
Compiler:	gcc (Red Hat 4.1.1-52)
RDB:	PostgreSQL 8.2.4

4.3 XPath Axis Determination Using Structural Information

Combining the code comparison method described in Section 4.2 and the operations introduced in Section 4.1, all the XPath axes listed in Table 1 can be determined. First, the descendant and following nodes can be obtained by extracting nodes from ranges **II** and **III**, respectively. Ancestor nodes can be obtained by using ***DO-3***, and then preceding nodes can be obtained by excluding ancestor nodes from range **I**. The parent node can be directly determined by ***DO-2***. Using the codes of the context node and its parent, the ranges **IV** and **V** for preceding-sibling and following-sibling nodes can be determined by using Algorithm 4, as shown in Figure 11. As the depth of preceding-sibling and following-sibling nodes are the same as the context node, and the child nodes are deeper by one than the context node, the preceding-sibling, following-sibling, and child nodes can be extracted from the ranges **IV**, **V** and **II** by using ***DO-1***. The extraction methods for XPath axes listed in Table 1 are summarized in Table 3.

5 Experimental Evaluation

We performed experiments within the environment described in Table 4 to evaluate the storage consumption and performance of ***DO-1***–***DO-3***, comparing the C-DO-VLEI code with C-ORDPATH. The XML documents used for the experiments were generated by xmlgen [3], using scale factors (SF) from 0.001 to 1, and were sourced from the *XML Data Repository* [2]. Table 5 shows the details of these XML documents. For the evaluation of performance, the labels of all

Table 5. XML documents

XML document	size (byte)	elements	max. depth	avg. depth	max. fanout	avg. fanout
SwissProt.xml	114820211	2977031	5	3.556711	50000	2.034511
dblp.xml	133862772	3332130	6	2.902279	328858	1.943555
ebay.xml	35525	156	5	3.75641	380	5.391305
item0.xml(xmlgen SF=1)	118552713	1666315	12	5.547796	25500	2.10692
item1.xml(xmlgen SF=0.1)	11875066	167865	12	5.548244	2550	2.103751
item2.xml(xmlgen SF=0.01)	1182547	17132	12	5.506012	255	2.102092
item3.xml(xmlgen SF=0.001)	118274	1729	12	5.717756	25	2.04965
lineitem.xml	32295475	1022976	3	2.941175	60175	1.941175
nasa.xml	25050288	476646	8	5.583141	2435	2.000728
orders.xml	5378845	150001	3	2.899987	15000	1.899987
part.xml	618181	20001	3	2.899905	2000	1.899905
partsupp.xml	2241868	48001	3	2.833295	8000	1.833295
psd7003.xml	716853012	21305818	7	5.15147	262526	1.818513
reed.xml	283619	10546	4	3.199791	703	1.809579
treebank_e.xml	86082517	2437666	36	7.872788	56384	1.571223
uwm.xml	2337522	66729	5	3.952435	2112	1.952276
wsu.xml	1647864	74557	4	3.157866	3924	2.077534

the nodes for each document were stored in a *document* table, and the labels of the nodes in all documents of the same depth were stored in a *depth* table.

We compared the C-DO-VLEI code with C-ORDPATH to evaluate the performance of ***DO-1***–***DO-3***[5]. Figure 12 shows the execution time ratio for extracting depth information, parent labels, and ancestor labels from the depth tables, using the C-DO-VLEI code and C-ORDPATH[6]. From this figure, we can see that the greater the node depth, the better is the performance by the C-DO-VLEI code in extracting node depth and parent labels. This is because delimiter detection in ORDPATH requires traversal from the head through the whole code and refers to the prefix schema for determining each delimiter. On the other hand, there is no significant difference in extracting the ancestor labels, because, even for the C-DO-VLEI code, output of all the ancestors requires handling of all delimiters one by one. Figure 13 shows the execution time ratio using the *document* tables. From these results, we deduce that the execution times for extracting depth information and parent labels using the C-DO-VLEI code are about 25% and 15% less than execution times using C-ORDPATH, respectively, which indicates that the proposed method can achieve high performance in such applications as XPath query and keyword search.

We then measured and compared the total label size of each XML document labeled by the C-DO-VLEI code and C-ORDPATH. Figure 14 shows the label size ratio of the C-DO-VLEI code to C-ORDPATH for each XML document, from which we can learn that the average label size using the C-DO-VLEI code is about 24% on the average smaller than that using the C-ORDPATH, which means taht the proposed method also outperforms the ORDPATH method with respect to storage consumption.

[5] There is no significant performance difference between C-DO-VLEI and C-ORDPATH in detecting other XPath axes, because they can determine such axes as descendant or following by simple code comparison. Therefore, only the experimental evaluation of ***DO-1***–***DO-3*** is discussed in this paper.

[6] Note that the label size using C-ORDPATH is set to 1 (same in Figure 13 and 14).

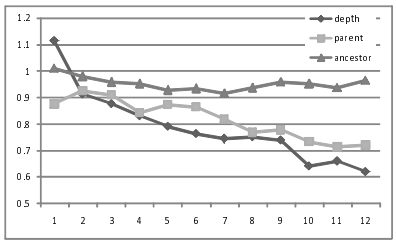

Fig. 12. Execution time ratio of C-DO-VLEI to C-ORDPATH (horizontal axis: depth of node)

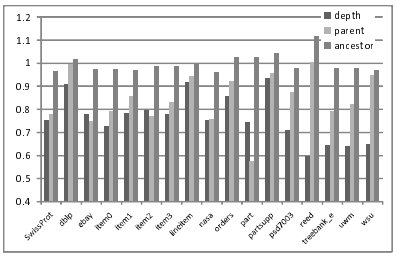

Fig. 13. Execution time ratio of C-DO-VLEI to C-ORDPATH (horizontal axis: XML document)

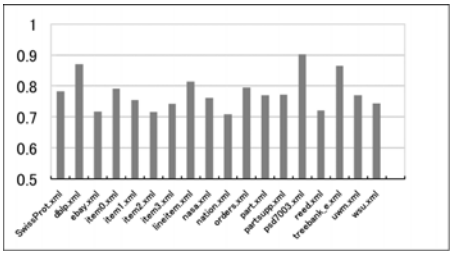

Fig. 14. Label size ratio of C-DO-VLEI to C-ORDPATH (horizontal axis: XML documents)

6 Conclusion

To achieve both efficient structural information extraction and low storage-consumption for large XML documents with high update frequency, in this paper we have proposed the C-DO-VLEI code, an update-friendly XML labeling method. To achieve efficient structural information extraction, we have proposed a novel and efficient method based on register-length bit operations combining with the properties of DO numbers, which are not able to be implemented in other bit-vector encoding schemes including ORDPATH. Meanwhile, the C-DO-VLEI code also achieves lower storage consumption by using a compressed binary representation using DO numbering schemes without any prefix schema.

We have performed experiments to evaluate and compare the storage consumption and performance of the proposed method with those of the C-ORDPATH method. Experimental results indicate that: 1) the execution times for extracting depth information and parent node labels using the C-DO-VLEI code are about 25% and 15% less, respectively, than for C-ORDPATH, which enables the proposed method to achieve high performance in such applications as XPath query and keyword search; 2) the label size using the C-DO-VLEI code is about 24%

smaller than that using C-ORDPATH, which means taht the proposed method also outperforms the ORDPATH method with respect to storage consumption.

References

1. MonetDB. http://monetdb.cwi.nl/.
2. XML Data Repository. http://www.cs.washington.edu/research/xmldatasets/.
3. Xmlgen. http://monetdb.cwi.nl/xml/downloads.html.
4. Toshiyuki Amagasa, Masatoshi Yoshikawa, and Shunsuke Uemura. QRS: A Robust Numbering Scheme for XML Documents. In *Proc. of ICDE*, pages 705–707, 2003.
5. Peter A. Boncz, Jan Flokstra, Torsten Grust, Maurice van Keulen, Stefan Manegold, K. Sjoerd Mullender, Jan Rittinger, and Jens Teubner. MonetDB/XQuery-Consistent and Efficient Updates on the Pre/Post Plane. In *Proc. of EDBT*, pages 1190–1193, 2006.
6. Peter A. Boncz, Torsten Grust, Maurice van Keulen, Stefan Manegold, Jan Rittinger, and Jens Teubner. MonetDB/XQuery: A Fast XQuery Processor Powered by a Relational Engine. In *Proc. of SIGMOD Conference*, pages 479–490, 2006.
7. Edith Cohen, Haim Kaplan, and Tova Milo. Labeling Dynamic XML Trees. In *Proc. of PODS*, pages 271–281, 2002.
8. Maggie Duong and Yanchun Zhang. LSDX: A New Labeling Scheme for Dynamically Updating XML Data. In *Proc. of ADC*, pages 185–193, 2005.
9. Alban Gabillon and Majirus Fansi. A persistent labelling scheme for XML and tree databases. In *Proc. of SITIS*, pages 110–115, 2005.
10. Dale Gerdemann. Parsing As Tree Traversal. In *Proc. of COLING*, pages 396–400, 1994.
11. Jr. Guy L Steele. *Hacker's Delight*. Addison-Wesley Professional, 2003.
12. Aye Khaing and Ni Lar Thein. A Persistent Labeling Scheme for Dynamic Ordered XML Trees. In *Proc. of Web Intelligence*, pages 498–501, 2006.
13. Kazuhito Kobayashi, Wenxin Liang, Dai Kobayashi, Akitsugu Watanabe, and Haruo Yokota. VLEI code: An Efficient Labeling Method for Handling XML Documents in an RDB. In *Proc. of ICDE*, pages 386–387, Tokyo, Japan, 2005(poster).
14. Changqing Li and Tok Wang Ling. Qed: a novel quaternary encoding to completely avoid re-labeling in xml updates. In *Proc. of CIKM*, pages 501–508, 2005.
15. Changqing Li, Tok Wang Ling, and Min Hu. Efficient Processing of Updates in Dynamic XML Data. In *Proc. of ICDE*, page 13, 2006.
16. Wenxin Liang, Takeshi Miki, and Haruo Yokota. Superimposed Code-based Indexing Method for Extracting MCTs from XML Documents. In *Proc. of DEXA*, pages 508–522, 2008.
17. Patrick E. O'Neil, Elizabeth J. O'Neil, Shankar Pal, Istvan Cseri, Gideon Schaller, and Nigel Westbury. ORDPATHs: Insert-Friendly XML Node Labels. In *Proc. of ACM SIGMOD Conference*, pages 903–908, 2004.
18. Virginie Sans and Dominique Laurent. Prefix Based Numbering Schemes for XML: Techniques, Applications and Performances. In *Proc. of VLDB*, pages 1564–1573, 2008.
19. Igor Tatarinov, Stratis Viglas, Kevin S. Beyer, Jayavel Shanmugasundaram, Eugene J. Shekita, and Chun Zhang. Storing and Querying Ordered XML Using a Relational Database System. In *Proc. of ACM SIGMOD Conference*, pages 204–215, 2002.
20. Xiaodong Wu, Mong-Li Lee, and Wynne Hsu. A Prime Number Labeling Scheme for Dynamic Ordered XML Trees. In *proc. of ICDE*, pages 66–78, 2004.